[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 9: Consistency and Consensus

Posted by CHARLES *on* 2020-04-18

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Is it better to be alive and wrong or right and dead? —Jay Kreps, A Few Notes on Kafka and Jepsen (2013)

- In this chapter, we will talk about some examples of algorithms and protocols for building **fault-tolerant distributed systems**.
- We will assume that all the problems from Chapter 8 can occur:
  - packets can be lost, reordered, duplicated,
  - arbitrarily delayed in the network;
  - clocks are approximate at best;
  - and nodes can pause (e.g., due to garbage collection) or crash at any time.
- The best way of building fault-tolerant systems is to find some **general-purpose abstractions with useful guarantees**, implement them once, and then let applications rely on those guarantees.
- This is the same approach as we used with transactions in Chapter 7: by using a transaction, the application can pretend:
  - that there are no crashes (**atomicity**),
  - that nobody else is concurrently accessing the database (**isolation**),
  - that storage devices are perfectly reliable (**durability**).
- One of the most important abstractions for distributed systems is **consensus**: that is, getting all of the nodes to agree on something.
  - Once you have an implementation of consensus, applications can use it for various purposes.
  - Correct implementations of consensus help avoid problems.

Consistency Guarantees (Weak vs. Strong)

name for eventual consistency may be **convergence**, as we expect all replicas to eventually converge to the same value.
  - **Weak guarantee**: it doesn't say anything about when the replicas will converge.
- **Eventual consistency** is hard for application developers because it is so different from the behavior of variables in a normal single-threaded program.
- Systems with **Stronger guarantees** may have worse performance or be less fault-tolerant than systems with weaker guarantees.
- **Distributed consistency models** vs. **Hierarchy of transaction isolation levels**
  - **transaction isolation** is primarily about avoiding race conditions due to concurrently executing transactions;
  - **distributed consistency** is mostly about coordinating the state of replicas in the face of delays and faults.

Linearizability(C: SLOW,not used very often in practice)

- Idea behind **Linearizability** (also known as **atomic consistency**, **strong consistency**, **immediate consistency**, or **external consistency**)
  - **The basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic**. (even though there may be multiple replicas in reality, the application does not need to worry about them.)
- **In a linearizable system**, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written.
  - In other words, linearizability is a **recency guarantee**.

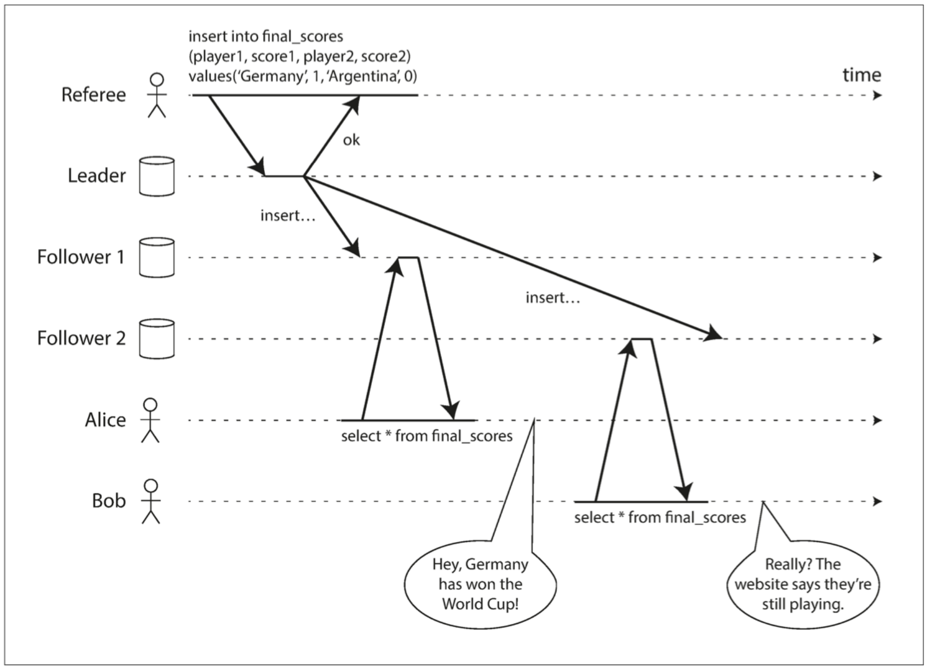
insert into final_scores
(player1, score1, player2, score2)
values('Germany', 1, 'Argentina', 0)
time
Referee
Leader
Follower 1
Follower 2
Alice
Bob
ok
insert...
insert...
select * from final_scores
select * from final_scores
Hey, Germany has won the World Cup!
Really? The website says they're still playing.

Figure 9-1. This system is not linearizable, causing football fans to be confused.

- **What Makes a System Linearizable**? To make a system appear as if there is only a single copy of the data.

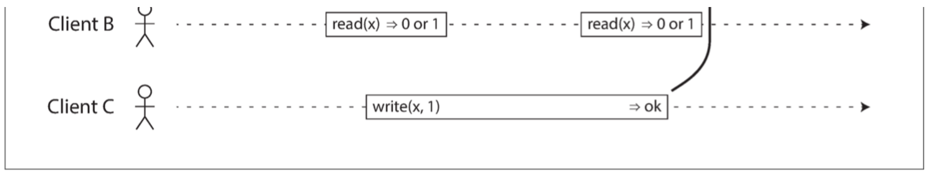

Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

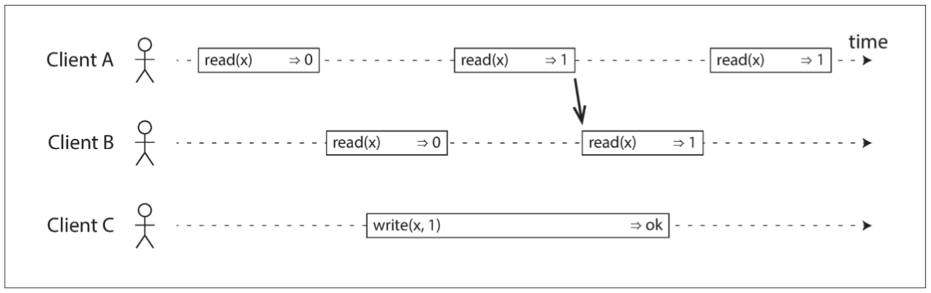

Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

- **atomic compare-and-set** (cas);

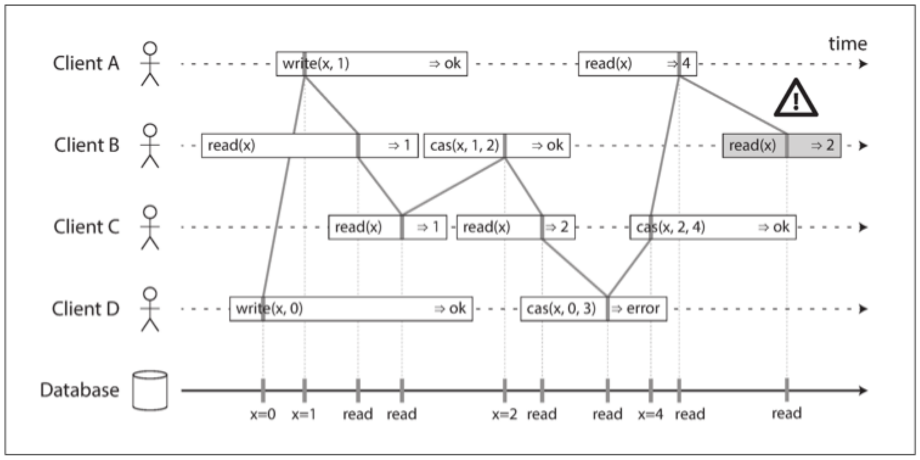

Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

- **Linearizability vs. Serializability**:
  - **Serializability** is an **isolation property of transactions**, where every transaction may read and write multiple objects (rows, documents, records).
  - **Linearizability** is a recency guarantee on reads and writes of a register (an individual object).
    - 2PL & actual serial execution are typically linearizable.
    - **serializable snapshot isolation** is **NOT** linearizable.
  - A database may provide both **serializability** and **linearizability**, and this combination is known as strict serializability or strong one-copy serializability.
- **Relying on Linearizability**:
  - In what circumstances is linearizability useful?
  - **Locking and leader election**: (e.g. ZooKeeper, etcd)

distributed databases, such as Oracle Real Application Clusters (RAC)

- **Constraints and Uniqueness guarantees**:
  - If you want to enforce this constraint as the data is written (such that if two people try to concurrently create a user or a file with the same name, one of them will be returned an error), you need **linearizability**. (C: kind like "lock")
  - a hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability.
- **Cross-channel timing dependencies**:
  - The linearizability violation was only noticed because there was an additional communication channel in the system.

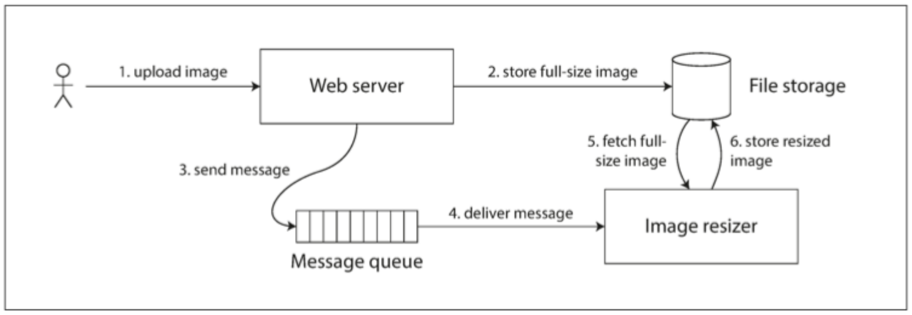

Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

- If the file storage service is linearizable, then this system should work fine.
- If it is not linearizable, there is the risk of a race condition:
  - the message queue (steps 3 and 4 in Figure 9-5) might be faster than the internal replication inside the storage service.
- This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue.
- **Implementing Linearizable Systems**:
  - **Replication & Linearizable**:
    - Single-leader replication (potentially linearizable);
    - Consensus algorithms (linearizable) [e.g. ZooKeeper and etcd]
    - Multi-leader replication (not linearizable)
    - Leaderless replication (probably not linearizable)
  - **Linearizability and quorums**:
    - it seems as though strict quorum reads and writes should be linearizable in a Dynamo-style model. However, when we have variable network delays, it is possible to have race conditions.

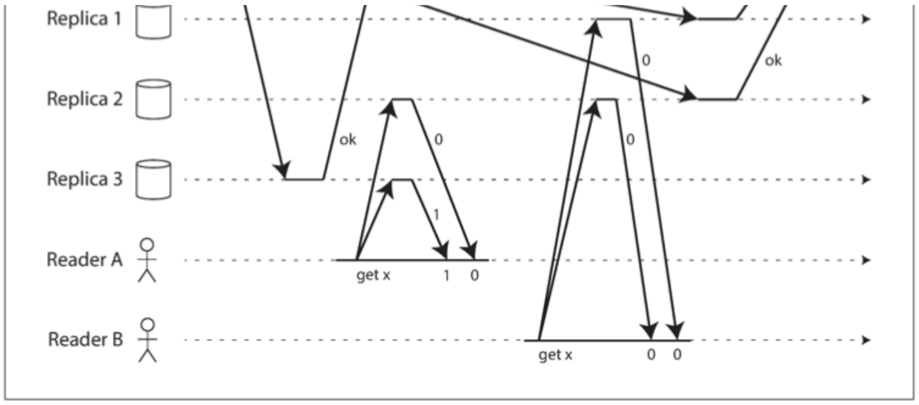

Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

- it is safest to assume that a leaderless system with Dynamo-style replication does not provide linearizability.
- **The Cost of Linearizability**:

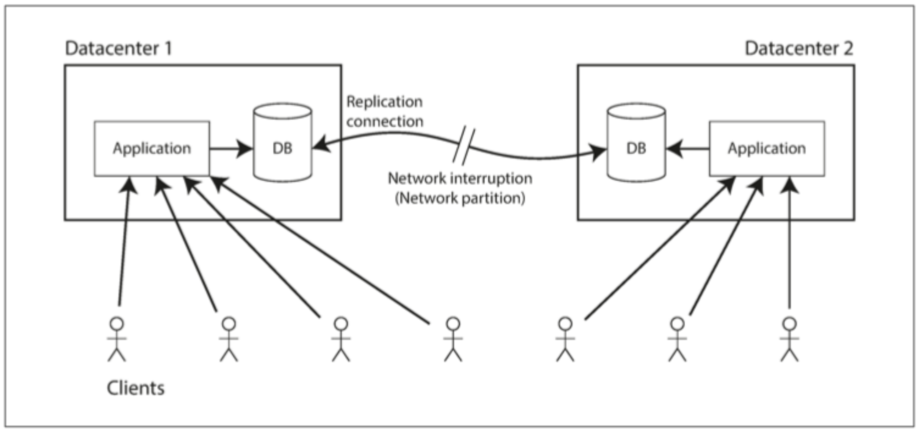

Figure 9-7. A network interruption forcing a choice between linearizability and availability.

- **The CAP theorem**: (very narrow scope)
  - CAP is sometimes presented as **Consistency**, **Availability**, **Partition tolerance**: pick 2 out of 3.
  - **The Unhelpful CAP Theorem**:
    - Unfortunately, putting it this way is misleading because **network partitions are a kind of fault**, so they aren't something about which you have a choice: **they will happen whether you like it or not**.
    - A better way of phrasing CAP would be either **Consistent** or **Available** when **Partitioned.** (aka, CP or AP)
    - All in all, there is a lot of misunderstanding and confusion around CAP, and **it does not help us understand systems better, so CAP is best avoided**.
- **Linearizability and network delays**:
  - Although linearizability is a useful guarantee, surprisingly few systems are actually linearizable in practice.

fault.

Ordering Guarantees

- Ordering has been a recurring theme in this book(e.g. Order of writes, Serializability, timestamps and clocks etc.)
- There are deep connections between **ordering**, **linearizability**, and **consensus**.
- **Ordering and Causality**:
  - There are several reasons why ordering keeps coming up, and one of the reasons is that it helps preserve **causality**.
    - E.g. causal dependency between the question and the answer;
  - **Causality imposes an ordering on events**: cause comes before effect;
    - a message is sent before that message is received;
    - the question comes before the answer.
  - If a system obeys the ordering imposed by causality, we say that it is **causally consistent.**
    - E.g. snapshot isolation provides causal consistency
  - **The causal order is NOT a total order**:
    - A **total order** allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller.
    - However, mathematical sets are not totally ordered: is {a, b} greater than {b, c}?
      - We say they are **incomparable**, and therefore mathematical sets are **partially ordered**.
    - **Linearizability**: In a linearizable system, we have a total order of operations.
    - **Causality**: two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. Hence **causality defines a partial order**.
  - There are **no concurrent operations in a linearizable datastore**: there must be a single timeline along which all operations are **totally ordered**.
  - **Distributed VCS** such as Git, their version histories are very much like the graph of causal dependencies.
    - Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.
  - **Linearizability is stronger than causal consistency**:
    - What is the relationship between the causal order and linearizability?
      - **linearizability implies causality**: any system that is linearizable will preserve causality correctly.
    - **Causal Consistency** is the **Strongest** possible consistency model that does **NOT slow down** due to network delays, and remains available in the face of network failures.

- **Capturing causal dependencies**:
    - In order to maintain causality, you need to know which operation happened before which other operation.
    - In order to determine causal dependencies, we need some way of describing the "knowledge" of a node in the system.
    - The techniques for determining which operation happened before which other operation are similar to what we discussed in "Detecting Concurrent Writes"
    - In order to determine the causal ordering, the database needs to know which version of the data was read by the application. (e.g. Version Vectors, or conflict detection of SSI)
- **Sequence Number Ordering**:
  - Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impractical.
  - Better way: we can use sequence numbers or timestamps to order events.
    - A **timestamp** need not come from a time-of-day clock but "**Logical Clock**".
    - Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a total order.
  - **Non-causal sequence number generators**:
    - If there is not a single leader (perhaps because you are using a multi-leader or leaderless database, or because the database is partitioned), it is less clear how to generate sequence numbers for operations.
      - Each node can generate its own independent set of sequence numbers. (e.g. one node odd, another node even; unique node identifier);
      - You can attach a timestamp from a time-of-day clock (physical clock) to each operation;
      - You can preallocate blocks of sequence numbers.
    - They all have a problem: **the sequence numbers they generate are not consistent with causality.**
  - **Lamport timestamps:**
    - a simple method for **generating sequence numbers that is consistent with causality**.
    - **The Lamport timestamp is then simply a pair of (counter, node ID)**.

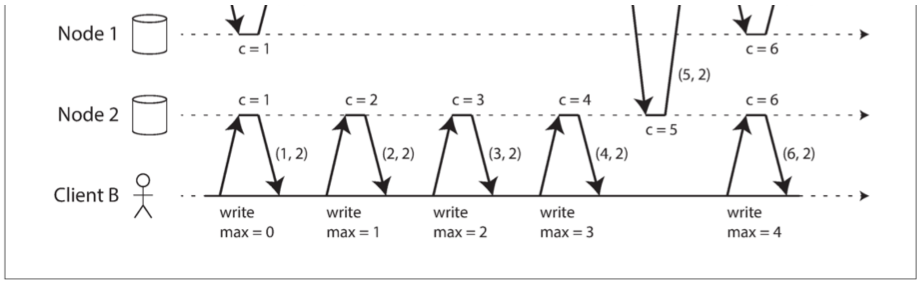

Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

- A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering:
  - if you have two timestamps, the one with a greater counter value is the greater timestamp;
  - if the counter values are the same, the one with the greater node ID is the greater timestamp.
- Key idea: **every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request**.
  - When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum. (C: kind like "Read Repair"?)
- As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp.
- **vs. Version Vector**:
  - version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other,
  - whereas Lamport timestamps always enforce a total ordering.
- **Timestamp ordering is not sufficient**:
  - The problem here is that the total order of operations only emerges after you have collected all of the operations.
  - To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized.
    - This idea of knowing when your total order is finalized is captured in the topic of **total order broadcast**.
- **Total Order Broadcast**:
  - If your program runs only on a single CPU core, it is easy to define a total ordering of operations: it is simply the order in which they were executed by the CPU. (but it is (no-surprising) tricky in distributed-system)
  - In the distributed systems literature, this problem is known as **total order broadcast or atomic broadcast**.
  - Total order broadcast → protocol for exchanging messages between nodes. With two safety properties:

order.

- **Using total order broadcast**: Log-style (e.g. Consensus services such as ZooKeeper and etcd)
  - there is a strong connection between **total order broadcast and consensus**.
  - Use cases:
    - Can be used for DB replication (aka. State machine replication)
    - Can be used to implement serializable transactions.
    - Useful for implementing a lock service that provides fencing tokens. (e.g. zxid in ZooKeeper)
  - **Total order broadcast** is that the order is fixed at the time the messages are delivered.
  - Another way of looking at total order broadcast is that it is a way of creating a log (as in a replication log, transaction log, or write-ahead log)
    - delivering a message is like appending to the log.
- **Implementing linearizable storage using total order broadcast**:
  - **Total order broadcast** is asynchronous: messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about when a message will be delivered (so one recipient may lag behind the others).
    - By contrast, **linearizability** is a recency guarantee: a read is guaranteed to see the latest value written.
  - While this procedure ensures **linearizable writes**, it doesn't guarantee linearizable reads. (due to async. update)
    - To be precise, the procedure described here provides **sequential consistency**, sometimes also known as **timeline consistency**, a slightly weaker guarantee than linearizability.
    - There are ways to make reads linearizable tho.
- **Implementing total order broadcast using linearizable storage**:
  - The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation.
    - Alternatively, an atomic compare-and-set operation would also do the job.
  - It can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both equivalent to consensus.

Distributed Transactions and Consensus

- **Consensus** is one of the most important and fundamental problems in distributed computing: the goal is simply to get several nodes to agree on something.
- Why is Consensus important ?
  - Leader election:
  - Atomic commit: aka. atomic commit problem.
- **The Impossibility of Consensus**: FLP-result with very restrictive rules.
- **Atomic Commit and Two-Phase Commit(2PC)**:

    - a node must only commit once it is certain that all other nodes in the transaction are also going to commit.
  - **Introduction to two-phase commit**:
    - Two-phase commit is an algorithm for **achieving atomic transaction** commit across multiple nodes.

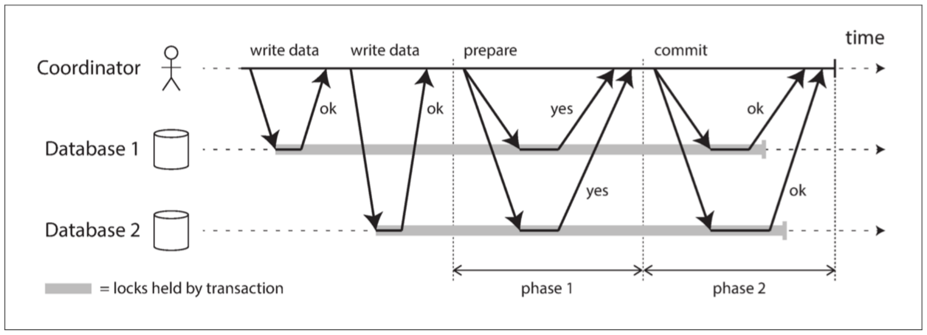

Figure 9-9. A successful execution of two-phase commit (2PC).

- 2PC uses a new component that does not normally appear in single-node transactions: a **coordinator** (aka. **transaction manager**).
- (This process is somewhat like the traditional marriage ceremony in Western cultures.)
- **A system of promises**:
  - There is no more going back: if the decision was to commit, that decision must be enforced, no matter how many retries it takes.
    - If a participant has crashed in the meantime, the transaction will be committed when it recovers—since the participant voted "yes," it cannot refuse to commit when it recovers.
- **Coordinator failure**:
  - If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction.
  - But once the participant has received a prepared request and voted "yes," it can no longer abort unilaterally—**it must wait to hear back from the coordinator**.
  - If the coordinator crashes or the network fails at this point, the participant can do nothing but wait.
    - A participant's transaction in this state is called **in doubt or uncertain**.

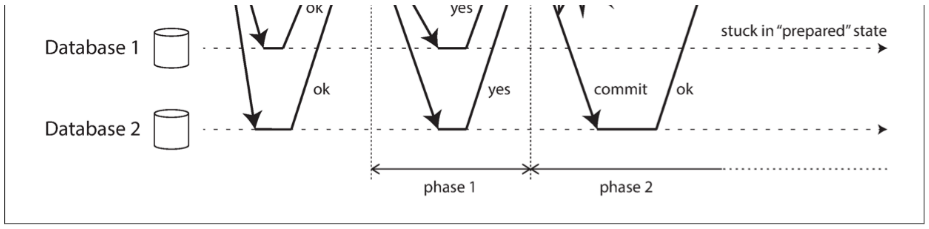

Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

- When the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log.
- Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.
- **Three-phase commit**:
  - Two-phase commit is called a blocking atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover.
  - 3PC assumes a network with bounded delay and nodes with bounded response times.
  - Non-blocking atomic commit requires a perfect failure detector.
- **Distributed Transactions in Practice**:
  - Many cloud services choose not to implement distributed transactions due to the operational problems they engender.
  - Some implementations of distributed transactions carry a heavy performance penalty
    - E.g. distributed transactions in MySQL are reported to be over 10 times slower than single-node transactions.
  - **Two quite different types of distributed transactions**:
    - **Database-internal distributed transactions**:
      - Some distributed databases (i.e., databases that use replication and partitioning in their standard configuration) support internal transactions among the nodes of that database.
    - **Heterogeneous distributed transactions**:
      - In a heterogeneous transaction, the participants are two or more different technologies: e.g. two databases from different vendors, or even non-database systems such as message brokers.
  - **Exactly-once message processing**:
    - Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways.
      - E.g. Message Queue work with DB.
    - by atomically committing the message and the side effects of its processing, we can ensure that the message is effectively processed exactly once, even if it required a few retries before it succeeded.
  - **XA transactions**: X/Open XA (short for eXtended Architecture)

transaction coordinator. (e.g. JTA → JDBC, JMS)

    - XA is supported by many traditional relational databases (including PostgreSQL, MySQL, DB2, SQL Server, and Oracle) and message brokers (including ActiveMQ, HornetQ, MSMQ, and IBM MQ).
  - **Holding locks while in doubt**:
    - database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes.
  - **Recovering from coordinator failure**:
    - In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions.
    - However, in practice, orphaned in-doubt transactions do occur — that is, transactions for which the coordinator cannot decide the outcome for whatever reason.
      - The only way out is for an administrator to manually decide whether to commit or roll back the transactions. (potentially requires a lot of manual effort)
    - Many XA implementations have an emergency escape hatch called **heuristic decisions**. (a euphemism for probably breaking atomicity)
  - **Limitations of distributed transactions**:
    - **Transaction coordinator** is itself a kind of database (in which transaction outcomes are stored), and so it needs to be approached with the same care as any other important database. (It like Single Point of Failure)
    - **Distributed transactions** thus have a **tendency of amplifying failures**, which runs counter to our goal of building fault-tolerant systems.
- **Fault-Tolerant Consensus**:
  - **Consensus problem**: one or more nodes may propose values, and the consensus algorithm decides on one of those values.
  - **Consensus algorithm must satisfy the following properties**: (Termination is a **liveness** property, whereas the other three are **safety** properties)
    - **Uniform agreement**:No two nodes decide differently.
    - **Integrity**:No node decides twice.
    - **Validity**:If a node decides value v, then v was proposed by some node.
    - **Termination**:Every node that does not crash eventually decides some value. (formalizes the idea of fault tolerance)
  - **Core idea**: everyone decides on the same outcome, and once you have decided, you cannot change your mind. (C: ALP 13, Disagree & Commit)
  - The **termination property** is subject to the assumption that fewer than half of the nodes are crashed or unreachable.
  - Most consensus algorithms assume that there are no Byzantine faults.
  - **Consensus algorithms and total order broadcast**
    - The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR), **Paxos**, Raft, and Zab;

- **Epoch numbering and quorums**:
  - All of the consensus protocols discussed so far internally use a leader in some form or another, but they don't guarantee that the leader is unique.
  - Instead, they can make a **weaker guarantee**:
    - The protocols define an **epoch number** (called the **ballot** number in Paxos, **view** number in Viewstamped Replication, and **term** number in Raft) and guarantee that within each epoch, the leader is unique.
  - How does a leader know that it hasn't been ousted by another node?
    - It must collect votes from a quorum of nodes.
    - The quorum typically, but not always, consists of a majority of nodes.
  - We have two rounds of voting: **once** to choose a leader, and a **second** time to vote on a leader's proposal.
    - **Key insight is** that the quorums for those two votes must overlap: if a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the most recent leader election.
  - vs. **2PC** which the coordinator is not elected.
    - Moreover, consensus algorithms define a recovery process by which nodes can get into a consistent state after a new leader is elected, ensuring that the safety properties are always met.
- **Limitations of consensus**:
  - Consensus algorithms are a huge breakthrough for distributed systems:
    - they bring concrete safety properties (agreement, integrity, and validity) to systems where everything else is uncertain,
    - they nevertheless remain fault-tolerant (able to make progress as long as a majority of nodes are working and reachable).
    - they can also implement linearizable atomic operations in a fault-tolerant way
  - But everything come at a cost:
  - Consensus systems **always require a strict majority to operate.**(1 out of 3, 2 out of 5 etc.)
  - Most consensus algorithms **assume** a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster.
  - Consensus systems generally rely on timeouts to detect failed nodes.(within highly variable network delays, this could be lots of false failed) result in frequent leader elections result in terrible performance.
  - Sometimes, consensus algorithms are particularly **sensitive to network problems**.

* **Membership and Coordination Services**:
  - Projects like **ZooKeeper** or **etcd** are often described as "distributed key-value stores" or "coordination and configuration services." (useful for distributed coordination)
  - ZooKeeper not for general-purpose databases, but indirectly via some other project.

entirely in memory (although they still write to disk for durability).

- That small amount of data is replicated across all the nodes using a **fault-tolerant total order broadcast** algorithm.

- **total order broadcast is just what you need for database replication**:
  - if each message represents a write to the database, applying the same writes in the same order keeps replicas consistent with each other.
- ZooKeeper not only **total order broadcast** (and hence consensus), but also:
  - **Linearizable atomic operations**: Using an atomic compare-and-set operation, you can implement a lock;
  - **Total ordering of operations**: fencing token -> monotonically increasing transaction ID (zxid) and version number (cversion);
  - **Failure detection**: the client and server periodically exchange heartbeats to check that the other node is still alive;
  - **Change notifications**: Not only can one client read locks and values that were created by another client, but it can also watch them for changes. (C: this featured used a lots for monitoring services)
- **Allocating work to nodes**:
  - Useful for single-leader databases, but it's also useful for job schedulers and similar stateful systems.
  - Partitioned resources (database, message streams, file storage, distributed actor system, etc.) and need to decide which partition to assign to which node.
  - If done correctly, this approach allows the application to automatically recover from faults without human intervention.
  - ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients.
  - If the application state needs to be replicated from one node to another, other tools (such as Apache BookKeeper) can be used.
- **Service discovery**:
  - **ZooKeeper**, **etcd**, and **Consul** are also often used for service discovery—that is, to find out which IP address you need to connect to in order to reach a particular service.
  - In cloud data center environments, where it is common for virtual machines to continually come and go, you often don't know the IP addresses of your services ahead of time.
    - Instead, you can configure your services such that when they start up they register their network endpoints in a service registry, where they can then be found by other services.
  - Some consensus systems support read-only caching replicas.
- **Membership services**:
  - **ZooKeeper** and **friends** can be seen as part of a long history of research into membership services.

Summary

- In this chapter we examined the topics of **consistency** and **consensus** from several different angles.
  - **Linearizability**(Consistency Model):
    - its goal is to make replicated data appear as though there were only a single copy, and to make all operations act on it atomically. (downside: being slow especially when there is large network delays)
  - **Causality**(weaker consistency model):
    - which imposes an ordering on events in a system (what happened before what, based on cause and effect).
    - **Causal consistency** does not have the coordination overhead of linearizability and is much less sensitive to network problems.
- We saw that achieving **consensus** means deciding something in such a way that all nodes agree on what was decided, and such that the decision is irrevocable.
  - Wide range of problems are actually reducible to consensus and are equivalent to each other:
    - **Linearizable compare-and-set registers**:The register needs to atomically decide whether to set its value, based on whether its current value equals the parameter given in the operation.
    - **Atomic transaction commit**:A database must decide whether to commit or abort a distributed transaction.
    - **Total order broadcast**:The messaging system must decide on the order in which to deliver messages.
    - **Locks and leases**:When several clients are racing to grab a lock or lease, the lock decides which one successfully acquired it.
    - **Membership/Coordination service**:Given a failure detector (e.g., timeouts), the system must decide which nodes are alive, and which should be considered dead because their sessions timed out.
    - **Uniqueness constraint**:When several transactions concurrently try to create conflicting records with the same key, the constraint must decide which one to allow and which should fail with a constraint violation.
  - How to handle leader-failure:
    - 1, Wait for the leader to recover, and accept that the system will be blocked in the meantime.
    - 2, Manually fail over by getting humans to choose a new leader node and reconfigure the system to use it.
    - 3, Use an algorithm to automatically choose a new leader.
- Tools like **ZooKeeper** play an important role in providing an “outsourced” **consensus**, **failure detection**, and **membership service** that applications can use.
- Not every system necessarily requires consensus: for example, **leaderless** and **multi-leader** replication systems typically do not use global consensus.